```
# ================================
# DJANGO BACKEND - AI TUTORING PLATFORM
# ================================

# ================================
# requirements.txt
# ================================
"""
Django==4.2.7
djangorestframework==3.14.0
django-cors-headers==4.3.1
django-extensions==3.2.3
python-decouple==3.8
psycopg2-binary==2.9.9
redis==5.0.1
celery==5.3.4
Pillow==10.1.0
PyPDF2==3.0.1
requests==2.31.0
google-generativeai==0.3.2
openai==1.3.7
serpapi==0.1.5
python-multipart==0.0.6
django-storages==1.14.2
boto3==1.34.0
whitenoise==6.6.0
gunicorn==21.2.0
"""

# ================================
# edugenius/settings.py
# ================================

import os
from decouple import config
from datetime import timedelta

BASE_DIR = Path(__file__).resolve().parent.parent

# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = config('SECRET_KEY', default='your-secret-key-here')

# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = config('DEBUG', default=True, cast=bool)

ALLOWED_HOSTS = config('ALLOWED_HOSTS', default='localhost,127.0.0.1', cast=lambda v:
[s.strip() for s in v.split(',')])

# Application definition
DJANGO_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]

THIRD_PARTY_APPS = [
    'rest_framework',
    'corsheaders',
    'django_extensions',
]

LOCAL_APPS = [
    'accounts',
    'learning',
    'quiz',
    'streaks',
    'ai_services',
]

INSTALLED_APPS = DJANGO_APPS + THIRD_PARTY_APPS + LOCAL_APPS

MIDDLEWARE = [
    'corsheaders.middleware.CorsMiddleware',
    'django.middleware.security.SecurityMiddleware',
    'whitenoise.middleware.WhiteNoiseMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]

ROOT_URLCONF = 'edugenius.urls'

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

```
WSGI_APPLICATION = 'edugenius.wsgi.application'

# Database
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': config('DB_NAME', default='edugenius'),
        'USER': config('DB_USER', default='postgres'),
        'PASSWORD': config('DB_PASSWORD', default='password'),
        'HOST': config('DB_HOST', default='localhost'),
        'PORT': config('DB_PORT', default='5432'),
    }
}

# Password validation
AUTH_PASSWORD_VALIDATORS = [
    {'NAME': 'django.contrib.auth.password_validation.UserAttributeSimilarityValidator'},
    {'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator'},
    {'NAME': 'django.contrib.auth.password_validation.CommonPasswordValidator'},
    {'NAME': 'django.contrib.auth.password_validation.NumericPasswordValidator'},
]

# Internationalization
LANGUAGE_CODE = 'en-us'
TIME_ZONE = 'UTC'
USE_I18N = True
USE_TZ = True

# Static files
STATIC_URL = '/static/'
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')

# Media files
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')

# Default primary key field type
DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'

# Custom user model
AUTH_USER_MODEL = 'accounts.User'

# REST Framework configuration
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.TokenAuthentication',
    ],
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated',
    ],
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 20,
}

# CORS settings
CORS_ALLOWED_ORIGINS = [
    "http://localhost:3000",
    "http://127.0.0.1:3000",
]

CORS_ALLOW_ALL_ORIGINS = DEBUG

# Redis configuration
REDIS_URL = config('REDIS_URL', default='redis://localhost:6379/0')

# Celery configuration
CELERY_BROKER_URL = REDIS_URL
CELERY_RESULT_BACKEND = REDIS_URL
CELERY_ACCEPT_CONTENT = ['json']
CELERY_TASK_SERIALIZER = 'json'
CELERY_RESULT_SERIALIZER = 'json'
CELERY_TIMEZONE = TIME_ZONE

# AI API Keys
GEMINI_API_KEY = config('GEMINI_API_KEY', default='')
OPENAI_API_KEY = config('OPENAI_API_KEY', default='')
SERP_API_KEY = config('SERP_API_KEY', default='')

# Cache configuration
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.redis.RedisCache',
        'LOCATION': REDIS_URL,
    }
}

# ================================
# accounts/models.py
# ================================

from django.contrib.auth.models import AbstractUser
from django.db import models
from django.utils import timezone

class User(AbstractUser):
    email = models.EmailField(unique=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['username']

class LearningProfile(models.Model):
    LEARNING_STYLES = [
        ('visual', 'Visual'),
        ('auditory', 'Auditory'),
        ('kinesthetic', 'Kinesthetic'),
        ('reading_writing', 'Reading/Writing'),
    ]

    LEARNING_PACE = [
        ('slow', 'Slow'),
        ('medium', 'Medium'),
        ('fast', 'Fast'),
    ]

    user = models.OneToOneField(User, on_delete=models.CASCADE)
    primary_learning_style = models.CharField(max_length=20, choices=LEARNING_STYLES,
null=True)
    learning_pace = models.CharField(max_length=10, choices=LEARNING_PACE,
default='medium')
    visual_score = models.IntegerField(default=0)
    auditory_score = models.IntegerField(default=0)
    kinesthetic_score = models.IntegerField(default=0)
    reading_writing_score = models.IntegerField(default=0)
    assessment_completed = models.BooleanField(default=False)
    assessment_date = models.DateTimeField(null=True, blank=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def save(self, *args, **kwargs):
        if not self.primary_learning_style and self.assessment_completed:
            scores = {
                'visual': self.visual_score,
                'auditory': self.auditory_score,
                'kinesthetic': self.kinesthetic_score,
                'reading_writing': self.reading_writing_score,
            }
            self.primary_learning_style = max(scores.keys(), key=scores.get)
        super().save(*args, **kwargs)

class LearningAssessment(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    answers = models.JSONField()
    completed_at = models.DateTimeField(auto_now_add=True)

# ================================
# learning/models.py
# ================================

from django.db import models
from django.contrib.auth import get_user_model

User = get_user_model()

class Subject(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()
    icon = models.URLField(blank=True)
    color = models.CharField(max_length=7, default='#3B82F6')  # Hex color
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.name

class Topic(models.Model):
    DIFFICULTY_LEVELS = [
        ('beginner', 'Beginner'),
        ('intermediate', 'Intermediate'),
        ('advanced', 'Advanced'),
    ]

    subject = models.ForeignKey(Subject, on_delete=models.CASCADE, related_name='topics')
    title = models.CharField(max_length=200)
    description = models.TextField()
    content = models.TextField()
    difficulty_level = models.CharField(max_length=20, choices=DIFFICULTY_LEVELS,
default='beginner')
    estimated_duration = models.IntegerField(help_text="Duration in minutes")
    order = models.PositiveIntegerField(default=0)
    is_active = models.BooleanField(default=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        ordering = ['order', 'created_at']

    def __str__(self):
        return f"{self.subject.name} - {self.title}"

class AdaptedContent(models.Model):
    topic = models.ForeignKey(Topic, on_delete=models.CASCADE,
related_name='adapted_contents')
    learning_style = models.CharField(max_length=20)
    content = models.TextField()
    visual_aids = models.JSONField(default=dict, blank=True)
    interactive_elements = models.JSONField(default=dict, blank=True)
    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        unique_together = ['topic', 'learning_style']

class StudySession(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    topic = models.ForeignKey(Topic, on_delete=models.CASCADE)
    start_time = models.DateTimeField(auto_now_add=True)
    end_time = models.DateTimeField(null=True, blank=True)
    completed = models.BooleanField(default=False)
    progress_percentage = models.FloatField(default=0.0)
    time_spent = models.DurationField(null=True, blank=True)

    def save(self, *args, **kwargs):
        if self.end_time and self.start_time:
            self.time_spent = self.end_time - self.start_time
        super().save(*args, **kwargs)

class UserTopicProgress(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    topic = models.ForeignKey(Topic, on_delete=models.CASCADE)
    progress_percentage = models.FloatField(default=0.0)
    completed = models.BooleanField(default=False)
    last_accessed = models.DateTimeField(auto_now=True)
    total_time_spent = models.DurationField(default=timezone.timedelta(0))

    class Meta:
        unique_together = ['user', 'topic']

class UploadedDocument(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    file = models.FileField(upload_to='documents/')
    file_type = models.CharField(max_length=10)  # pdf, txt, docx
    processed = models.BooleanField(default=False)
    content_extracted = models.TextField(blank=True)
    created_at = models.DateTimeField(auto_now_add=True)

# ================================
# quiz/models.py
# ================================

from django.db import models
from django.contrib.auth import get_user_model
from learning.models import Topic

User = get_user_model()

class Quiz(models.Model):
    topic = models.ForeignKey(Topic, on_delete=models.CASCADE, related_name='quizzes')
    title = models.CharField(max_length=200)
    description = models.TextField(blank=True)
    questions = models.JSONField()  # Store questions as JSON
    passing_score = models.IntegerField(default=75)
    time_limit = models.IntegerField(null=True, blank=True, help_text="Time limit in minutes")
    is_active = models.BooleanField(default=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return f"Quiz: {self.title}"
```

```
class QuizAttempt(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    quiz = models.ForeignKey(Quiz, on_delete=models.CASCADE)
    answers = models.JSONField()
    score = models.FloatField()
    passed = models.BooleanField()
    time_taken = models.DurationField(null=True, blank=True)
    attempt_number = models.PositiveIntegerField()
    started_at = models.DateTimeField()
    completed_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        unique_together = ['user', 'quiz', 'attempt_number']
        ordering = ['-completed_at']

class QuizQuestion(models.Model):
    QUESTION_TYPES = [
        ('multiple_choice', 'Multiple Choice'),
        ('true_false', 'True/False'),
        ('short_answer', 'Short Answer'),
        ('essay', 'Essay'),
    ]

    quiz = models.ForeignKey(Quiz, on_delete=models.CASCADE, related_name='quiz_questions')
    question_text = models.TextField()
    question_type = models.CharField(max_length=20, choices=QUESTION_TYPES)
    options = models.JSONField(default=dict, blank=True)
    correct_answer = models.TextField()
    explanation = models.TextField(blank=True)
    points = models.FloatField(default=1.0)
    order = models.PositiveIntegerField(default=0)

    class Meta:
        ordering = ['order']

# ================================
# streaks/models.py
# ================================

from django.db import models
from django.contrib.auth import get_user_model
from django.utils import timezone

User = get_user_model()

class StreakType(models.Model):
    name = models.CharField(max_length=50)
    description = models.TextField()
    icon = models.CharField(max_length=50)  # emoji or icon class
    points_per_day = models.IntegerField(default=10)
    color = models.CharField(max_length=7, default='#FF6B6B')
    def __str__(self):
        return self.name

class UserStreak(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    streak_type = models.ForeignKey(StreakType, on_delete=models.CASCADE)
    current_streak = models.IntegerField(default=0)
    longest_streak = models.IntegerField(default=0)
    total_points = models.IntegerField(default=0)
    last_activity_date = models.DateField()
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        unique_together = ['user', 'streak_type']

    def update_streak(self):
        today = timezone.now().date()
        yesterday = today - timezone.timedelta(days=1)

        if self.last_activity_date == yesterday:
            # Continue streak
            self.current_streak += 1
        elif self.last_activity_date == today:
            # Already updated today
            return
        else:
            # Streak broken, reset
            self.current_streak = 1

        if self.current_streak > self.longest_streak:
            self.longest_streak = self.current_streak

        self.total_points += self.streak_type.points_per_day
        self.last_activity_date = today
        self.save()

class Achievement(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()
    icon = models.CharField(max_length=50)
    condition = models.JSONField()  # Condition for unlocking
    points = models.IntegerField(default=0)
    badge_color = models.CharField(max_length=7, default='#FFD700')
    is_active = models.BooleanField(default=True)

    def __str__(self):
        return self.name

class UserAchievement(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    achievement = models.ForeignKey(Achievement, on_delete=models.CASCADE)
    earned_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        unique_together = ['user', 'achievement']

# ==================================
# ai_services/services.py
# ==================================

import google.generativeai as genai
import openai
import requests
from serpapi import GoogleSearch
from django.conf import settings
from django.core.cache import cache
import json
import logging

logger = logging.getLogger(__name__)

class AIService:
    def __init__(self):
        # Configure AI APIs
        genai.configure(api_key=settings.GEMINI_API_KEY)
        openai.api_key = settings.OPENAI_API_KEY
        self.gemini_model = genai.GenerativeModel('gemini-1.5-pro')

    def analyze_pdf_content(self, file_path, learning_style):
        """Extract and analyze PDF content"""
        cache_key = f"pdf_analysis_{hash(file_path)}_{learning_style}"
        cached_result = cache.get(cache_key)
        if cached_result:
            return cached_result

        try:
            with open(file_path, 'rb') as file:
                prompt = f"""
                Analyze this educational document and create learning content optimized for a
{learning_style} learner.

                For Visual learners: Focus on creating descriptions for diagrams, charts, visual patterns,
and suggest infographics.
                For Auditory learners: Focus on creating discussion points, verbal explanations, and
audio-friendly content.
                For Kinesthetic learners: Focus on hands-on activities, practical applications, and
interactive elements.
                For Reading/Writing learners: Focus on detailed text, summaries, key points, and
structured notes.

                Extract key concepts, create section summaries, and identify main topics.
                Return as JSON with structure:
                {{
```

```
            "title": "Document title",
            "summary": "Brief summary",
            "sections": [
                {{
                    "heading": "Section title",
                    "content": "Adapted content for {learning_style} learner",
                    "key_points": ["point1", "point2"],
                    "suggested_activities": ["activity1", "activity2"]
                }}
            ],
            "quiz_topics": ["topic1", "topic2", "topic3"]
        }}
        """

        response = self.gemini_model.generate_content([prompt, file])
        result = self.parse_json_response(response.text)
        cache.set(cache_key, result, 3600)  # Cache for 1 hour
        return result

    except Exception as e:
        logger.error(f"Error analyzing PDF: {str(e)}")
        return None

def generate_adapted_content(self, topic, raw_content, learning_style):
    """Generate learning style adapted content"""
    cache_key = f"adapted_content_{hash(topic)}_{learning_style}"
    cached_result = cache.get(cache_key)
    if cached_result:
        return cached_result

    style_prompts = {
        'visual': """
        Create visual learning content with:
        1. Descriptions of diagrams and charts needed
        2. Color-coded information organization
        3. Visual memory aids and mnemonics
        4. Infographic suggestions
        5. Mind map structures
        """,
        'auditory': """
        Create auditory learning content with:
        1. Discussion questions and talking points
        2. Verbal explanations and analogies
        3. Rhythmic or musical memory aids
        4. Group discussion activities
        5. Read-aloud friendly formatting
        """,
        'kinesthetic': """
        Create hands-on learning content with:
        1. Interactive exercises and activities
        2. Real-world applications and examples
        3. Step-by-step practical procedures
```

```
            4. Movement-based learning activities
            5. Simulation and role-play scenarios
            """,
            'reading_writing': """
            Create text-based learning content with:
            1. Detailed written explanations
            2. Structured notes and outlines
            3. Lists and bullet points
            4. Writing exercises and reflections
            5. Research and documentation tasks
            """
        }

        try:
            prompt = f"""
            Topic: {topic}
            Original Content: {raw_content}

            {style_prompts.get(learning_style, style_prompts['visual'])}

            Create comprehensive learning content adapted for {learning_style} learners.
            Include interactive elements and engagement strategies.

            Return as JSON:
            {{
                "adapted_content": "Main content adapted for learning style",
                "activities": ["activity1", "activity2"],
                "visual_aids": ["aid1", "aid2"],
                "assessment_questions": ["q1", "q2", "q3"]
            }}
            """

            response = openai.ChatCompletion.create(
                model="gpt-3.5-turbo",
                messages=[{"role": "user", "content": prompt}],
                max_tokens=1500
            )

            result = self.parse_json_response(response.choices[0].message.content)
            cache.set(cache_key, result, 3600)
            return result

        except Exception as e:
            logger.error(f"Error generating adapted content: {str(e)}")
            return None

    def generate_quiz(self, topic_content, difficulty_level="medium", num_questions=5):
        """Generate quiz questions"""
        try:
            prompt = f"""
            Generate {num_questions} comprehensive quiz questions for the following content:
```

```
        {topic_content}

        Requirements:
        - Difficulty: {difficulty_level}
        - Mix of question types: multiple choice, true/false, short answer
        - Test understanding, not just memorization
        - Include detailed explanations for correct answers

        Return as JSON:
        {{
            "questions": [
                {{
                    "question": "Question text",
                    "type": "multiple_choice",
                    "options": ["Option A", "Option B", "Option C", "Option D"],
                    "correct_answer": "Option B",
                    "explanation": "Detailed explanation why this is correct",
                    "points": 20
                }}
            ]
        }}
        """

        response = self.gemini_model.generate_content(prompt)
        return self.parse_json_response(response.text)

    except Exception as e:
        logger.error(f"Error generating quiz: {str(e)}")
        return None

def search_educational_content(self, topic, learning_style):
    """Search for educational content online"""
    try:
        params = {
            "engine": "google",
            "q": f"{topic} educational content {learning_style} learning resources",
            "api_key": settings.SERP_API_KEY,
            "num": 10
        }

        search = GoogleSearch(params)
        results = search.get_dict()

        if "organic_results" in results:
            processed_results = []
            for result in results["organic_results"][:5]:
                processed_results.append({
                    "title": result.get("title", ""),
                    "link": result.get("link", ""),
                    "snippet": result.get("snippet", ""),
                })
            return processed_results
        except Exception as e:
            logger.error(f"Error searching content: {str(e)}")

        return []

    def evaluate_quiz_answer(self, question, student_answer, correct_answer, question_type):
        """Evaluate student answer using AI"""
        if question_type == "multiple_choice" or question_type == "true_false":
            return {
                "score": 100 if student_answer.lower() == correct_answer.lower() else 0,
                "feedback": "Correct!" if student_answer.lower() == correct_answer.lower() else
f"Incorrect. The correct answer is: {correct_answer}"
            }

        try:
            prompt = f"""
            Evaluate this student answer:
            Question: {question}
            Student Answer: {student_answer}
            Correct Answer: {correct_answer}

            Provide a score (0-100) and constructive feedback.
            Be generous with partial credit for partially correct answers.

            Return as JSON:
            {{
                "score": 85,
                "feedback": "Good understanding but missed key point about..."
            }}
            """

            response = self.gemini_model.generate_content(prompt)
            return self.parse_json_response(response.text)

        except Exception as e:
            logger.error(f"Error evaluating answer: {str(e)}")
            return {"score": 0, "feedback": "Unable to evaluate answer"}

    def parse_json_response(self, response_text):
        """Parse JSON from AI response, handling markdown formatting"""
        try:
            # Remove markdown code blocks if present
            cleaned_text = response_text.replace("```json", "").replace("```", "").strip()
            return json.loads(cleaned_text)
        except json.JSONDecodeError:
            logger.error(f"Failed to parse JSON: {response_text}")
            return {}

# ================================
# API Views and Serializers
# ================================
```

```
# accounts/serializers.py
from rest_framework import serializers
from django.contrib.auth import get_user_model
from .models import LearningProfile, LearningAssessment

User = get_user_model()

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ['id', 'username', 'email', 'first_name', 'last_name', 'created_at']
        read_only_fields = ['id', 'created_at']

class LearningProfileSerializer(serializers.ModelSerializer):
    class Meta:
        model = LearningProfile
        fields = '__all__'
        read_only_fields = ['user', 'created_at', 'updated_at']

class LearningAssessmentSerializer(serializers.ModelSerializer):
    class Meta:
        model = LearningAssessment
        fields = ['answers', 'completed_at']

# accounts/views.py
from rest_framework import generics, status
from rest_framework.decorators import api_view, permission_classes
from rest_framework.permissions import IsAuthenticated
from rest_framework.response import Response
from django.contrib.auth import authenticate
from rest_framework.authtoken.models import Token
from .models import LearningProfile
from .serializers import UserSerializer, LearningProfileSerializer, LearningAssessmentSerializer

@api_view(['POST'])
@permission_classes([])
def register_user(request):
    serializer = UserSerializer(data=request.data)
    if serializer.is_valid():
        user = serializer.save()
        user.set_password(request.data['password'])
        user.save()

        # Create learning profile
        LearningProfile.objects.create(user=user)

        token, created = Token.objects.get_or_create(user=user)
        return Response({
            'token': token.key,
            'user': UserSerializer(user).data
        }, status=status.HTTP_201_CREATED)
```

```python
    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

@api_view(['POST'])
@permission_classes([])
def login_user(request):
    email = request.data.get('email')
    password = request.data.get('password')

    user = authenticate(username=email, password=password)
    if user:
        token, created = Token.objects.get_or_create(user=user)
        return Response({
            'token': token.key,
            'user': UserSerializer(user).data
        })
    return Response({'error': 'Invalid credentials'}, status=status.HTTP_401_UNAUTHORIZED)

@api_view(['POST'])
@permission_classes([IsAuthenticated])
def submit_learning_assessment(request):
    """Submit learning style assessment"""
    answers = request.data.get('answers', [])

    # Calculate scores based on answers
    visual_score = sum(1 for answer in answers if answer.get('style') == 'visual')
    auditory_score = sum(1 for answer in answers if answer.get('style') == 'auditory')
    kinesthetic_score = sum(1 for answer in answers if answer.get('style') == 'kinesthetic')
    reading_writing_score = sum(1 for answer in answers if answer.get('style') == 'reading_writing')

    # Update learning profile
    profile = request.user.learningprofile
    profile.visual_score = visual_score
    profile.auditory_score = auditory_score
    profile.kinesthetic_score = kinesthetic_score
    profile.reading_writing_score = reading_writing_score
    profile.assessment_completed = True
    profile.assessment_date = timezone.now()
    profile.save()

    return Response(LearningProfileSerializer(profile).data)

# learning/serializers.py
from rest_framework import serializers
from .models import Subject, Topic, StudySession, UserTopicProgress, UploadedDocument

class SubjectSerializer(serializers.ModelSerializer):
    class Meta:
        model = Subject
        fields = '__all__'

class TopicSerializer(serializers.ModelSerializer):
    subject_name = serializers.CharField(source='subject.name', read_only=True)
    class Meta:
        model = Topic
        fields = '__all__'

class StudySessionSerializer(serializers.ModelSerializer):
    class Meta:
        model = StudySession
        fields = '__all__'
        read_only_fields = ['user']

class UserTopicProgressSerializer(serializers.ModelSerializer):
    topic_title = serializers.CharField(source='topic.title', read_only=True)

    class Meta:
        model = UserTopicProgress
        fields = '__all__'
        read_only_fields = ['user']

# learning/views.py
from rest_framework import viewsets, status
from rest_framework.decorators import action
from rest_framework.response import Response
from rest_framework.permissions import IsAuthenticated
from django.shortcuts import get_object_or_404
from .models import Subject, Topic, StudySession, UserTopicProgress, UploadedDocument
from .serializers import SubjectSerializer, TopicSerializer, StudySessionSerializer
from ai_services.services import AIService
import os

class SubjectViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = Subject.objects.all()
    serializer_class = SubjectSerializer
    permission_classes = [IsAuthenticated]

class TopicViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = Topic.objects.filter(is_active=True)
    serializer_class = TopicSerializer
    permission_classes = [IsAuthenticated]

    @action(detail=True, methods=['get'])
    def content(self, request, pk=None):
        """Get adapted content for user's learning style"""
        topic = self.get_object()
        user_learning_style = request.user.learningprofile.primary_learning_style

        if not user_learning_style:
            return Response({'error': 'Please complete learning style assessment first'},
                          status=status.HTTP_400_BAD_REQUEST)

        # Try to get existing adapted content
        adapted_content = topic.adapted_contents.filter(learning_style=user_learning_style).first()
        if not adapted_content:
            # Generate new adapted content
            ai_service = AIService()
            content_data = ai_service.generate_adapted_content(
                topic.title,
                topic.content,
                user_learning_style
            )

            if content_data:
                from learning.models import AdaptedContent
                adapted_content = AdaptedContent.objects.create(
                    topic=topic,
                    learning_style=user_learning_style,
                    content=content_data.get('adapted_content', ''),
                    visual_aids=content_data.get('visual_aids', []),
                    interactive_elements=content_data.get('activities', [])
                )

        return Response({
            'topic': TopicSerializer(topic).data,
            'adapted_content': adapted_content.content if adapted_content else topic.content,
            'visual_aids': adapted_content.visual_aids if adapted_content else [],
            'interactive_elements': adapted_content.interactive_elements if adapted_content else []
        })

    @action(detail=True, methods=['post'])
    def start_session(self, request, pk=None):
        """Start a study session for a topic"""
        topic = self.get_object()
        session = StudySession.objects.create(
            user=request.user,
            topic=topic
        )
        return Response(StudySessionSerializer(session).data)

    @action(detail=True, methods=['patch'])
    def update_progress(self, request, pk=None):
        """Update user's progress on a topic"""
        topic = self.get_object()
        progress_percentage = request.data.get('progress_percentage', 0)
        completed = request.data.get('completed', False)

        progress, created = UserTopicProgress.objects.get_or_create(
            user=request.user,
            topic=topic,
            defaults={'progress_percentage': progress_percentage, 'completed': completed}
        )

        if not created:
            progress.progress_percentage = progress_percentage
            progress.completed = completed
            progress.save()

        return Response({'progress': progress_percentage, 'completed': completed})

@api_view(['POST'])
@permission_classes([IsAuthenticated])
def upload_document(request):
    """Upload and process PDF documents"""
    if 'file' not in request.FILES:
        return Response({'error': 'No file provided'}, status=status.HTTP_400_BAD_REQUEST)

    file = request.FILES['file']
    title = request.data.get('title', file.name)

    # Save uploaded document
    document = UploadedDocument.objects.create(
        user=request.user,
        title=title,
        file=file,
        file_type=file.name.split('.')[-1].lower()
    )

    # Process document with AI
    ai_service = AIService()
    user_learning_style = request.user.learningprofile.primary_learning_style

    if user_learning_style:
        result = ai_service.analyze_pdf_content(document.file.path, user_learning_style)
        if result:
            document.content_extracted = json.dumps(result)
            document.processed = True
            document.save()

            return Response({
                'document_id': document.id,
                'processed_content': result
            })

    return Response({'error': 'Failed to process document'},
status=status.HTTP_500_INTERNAL_SERVER_ERROR)

# quiz/serializers.py
from rest_framework import serializers
from .models import Quiz, QuizAttempt

class QuizSerializer(serializers.ModelSerializer):
    class Meta:
        model = Quiz
        fields = '__all__'

class QuizAttemptSerializer(serializers.ModelSerializer):
    class Meta:
        model = QuizAttempt
        fields = '__all__'
        read_only_fields = ['user']

# quiz/views.py
from rest_framework import viewsets, status
from rest_framework.decorators import action
from rest_framework.response import Response
from rest_framework.permissions import IsAuthenticated
from django.utils import timezone
from .models import Quiz, QuizAttempt
from .serializers import QuizSerializer, QuizAttemptSerializer
from ai_services.services import AIService

class QuizViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = Quiz.objects.filter(is_active=True)
    serializer_class = QuizSerializer
    permission_classes = [IsAuthenticated]

    @action(detail=True, methods=['post'])
    def submit(self, request, pk=None):
        """Submit quiz answers for grading"""
        quiz = self.get_object()
        answers = request.data.get('answers', {})
        started_at = request.data.get('started_at')

        if started_at:
            started_at = timezone.datetime.fromisoformat(started_at.replace('Z', '+00:00'))
        else:
            started_at = timezone.now()

        # Get attempt number
        attempt_number = QuizAttempt.objects.filter(user=request.user, quiz=quiz).count() + 1

        # Grade the quiz
        ai_service = AIService()
        total_score = 0
        total_possible = 0
        detailed_results = []

        questions = quiz.questions.get('questions', [])
        for i, question in enumerate(questions):
            user_answer = answers.get(str(i), '')
            question_type = question.get('type', 'multiple_choice')
            correct_answer = question.get('correct_answer', '')
            points = question.get('points', 20)

            result = ai_service.evaluate_quiz_answer(
                question['question'],
                user_answer,
                correct_answer,
                question_type
            )

            score = result['score'] * (points / 100)
            total_score += score
            total_possible += points

            detailed_results.append({
                'question': question['question'],
                'user_answer': user_answer,
                'correct_answer': correct_answer,
                'score': score,
                'max_points': points,
                'feedback': result['feedback']
            })

        # Calculate final percentage
        final_percentage = (total_score / total_possible * 100) if total_possible > 0 else 0
        passed = final_percentage >= quiz.passing_score

        # Create quiz attempt record
        attempt = QuizAttempt.objects.create(
            user=request.user,
            quiz=quiz,
            answers=answers,
            score=final_percentage,
            passed=passed,
            attempt_number=attempt_number,
            started_at=started_at,
            time_taken=timezone.now() - started_at
        )

        return Response({
            'attempt_id': attempt.id,
            'score': final_percentage,
            'passed': passed,
            'passing_score': quiz.passing_score,
            'detailed_results': detailed_results,
            'can_retake': not passed
        })

# URL Configuration
# edugenius/urls.py
from django.contrib import admin
from django.urls import path, include
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/auth/', include('accounts.urls')),
    path('api/learning/', include('learning.urls')),
    path('api/quiz/', include('quiz.urls')),
    path('api/streaks/', include('streaks.urls')),
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)

# accounts/urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('register/', views.register_user, name='register'),
    path('login/', views.login_user, name='login'),
    path('assessment/', views.submit_learning_assessment, name='learning_assessment'),
]

# learning/urls.py
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from . import views

router = DefaultRouter()
router.register(r'subjects', views.SubjectViewSet)
router.register(r'topics', views.TopicViewSet)

urlpatterns = [
    path('', include(router.urls)),
    path('upload-document/', views.upload_document, name='upload_document'),
]

# quiz/urls.py
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from . import views

router = DefaultRouter()
router.register(r'quizzes', views.QuizViewSet)

urlpatterns = [
    path('', include(router.urls)),
]

# Celery configuration
# celery.py
import os
from celery import Celery

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'edugenius.settings')

app = Celery('edugenius')
app.config_from_object('django.conf:settings', namespace='CELERY')
app.autodiscover_tasks()

# tasks.py (in ai_services app)
```

```
from celery import shared_task
from .services import AIService
import logging

logger = logging.getLogger(__name__)

@shared_task
def generate_quiz_for_topic(topic_id, learning_style):
    """Background task to generate quiz questions"""
    try:
        from learning.models import Topic
        from quiz.models import Quiz

        topic = Topic.objects.get(id=topic_id)
        ai_service = AIService()

        quiz_data = ai_service.generate_quiz(topic.content, "medium", 5)

        if quiz_data:
            Quiz.objects.create(
                topic=topic,
                title=f"Quiz: {topic.title}",
                questions=quiz_data,
                passing_score=75
            )
            logger.info(f"Quiz generated for topic {topic_id}")

    except Exception as e:
        logger.error(f"Failed to generate quiz for topic {topic_id}: {str(e)}")

@shared_task
def process_uploaded_document(document_id):
    """Background task to process uploaded documents"""
    try:
        from learning.models import UploadedDocument

        document = UploadedDocument.objects.get(id=document_id)
        ai_service = AIService()

        # Get user's learning style
        learning_style = document.user.learningprofile.primary_learning_style

        if learning_style:
            result = ai_service.analyze_pdf_content(document.file.path, learning_style)
            if result:
                document.content_extracted = json.dumps(result)
                document.processed = True
                document.save()
                logger.info(f"Document {document_id} processed successfully")

    except Exception as e:
        logger.error(f"Failed to process document {document_id}: {str(e)}")
```